GENERAL TECHNIQUES FOR AUTOMATIC PROGRAM OPTIMIZATION AND SYNTHESIS THROUGH THEOREM PROVING

Peter MADDEN and Alan BUNDY

DAI Research Paper No. 644
1993

To appear in the Proceedings of EWAIC'93

General Techniques for Automatic Program Optimization and Synthesis Through Theorem Proving[1]

To appear in *Proceedings of:*
EAST-WEST I CONFERENCE: from theory to practice EWAIC'93
Moscow, September 7-9, 1993

Peter Madden and Alan Bundy
Department of AI, University of Edinburgh,
80 South Bridge, Edinburgh EH1 1HN, Scotland.
Tel: *+44 31 650 2722*
E-mail: *{pm,bundy}@uk.ac.ed.aisb*
fax: *+44 31 650 6516*

Abstract

We report on program optimization research within the *proofs as programs* paradigm. Firstly, we describe program optimization by the transformation of program synthesis proofs (i.e. constructive existence proofs). Synthesis proofs which yield inefficient programs are transformed into analogous proofs which yield more efficient programs. The key to program optimization lies in the transformation of the various induction schemas employed in synthesis proofs. This belief stems from the extensive work of Boyer and Moore concerning the dualities between induction and recursion. A system has been implemented which optimizes simple recursive behaviour by automatically transforming the associated synthesis proof structures (notably, the forms of mathematical induction used to synthesize recursive program constucts).

A second approach to program optimization does not concern how target synthesis proofs can be "mapped" from source proofs. Rather, the question is can we devise a means by which these proofs can be automatically constructed, *without* the use of a source proof (but only the source equational definitions). A promising strategy is to use the "proof planning" approach [Bundy *et al*, 1991b] to theorem proving and to employ *meta-variables* at the meta-level planning phase which allow the planning to proceed even though certain object-level objects are (partially) unknown (such a strategy being known as *middle-out reasoning* [Bundy *et al*, 1990a]). Subsequent planning provides the necessary information which, together with the original definitional equations, will allow us to instantiate such meta-variables through *higher-order unification* procedures. The control provided by proof planning allows us to view program synthesis as verification together with MOR. This approach has already been investigated for the purposes of synthesizing tail-recursive programs from non-tail-recursive specifications [Hesketh *et al*, 1992],

[1]This research was supported by SERC grant GR/H/23610, and a SERC Postdoctoral Fellowship to the author.

as well as other forms of program optimization such as tupling transformations and deforestation [Madden *et al*, 1993a, Wiggins *et al*, 1991].

This research has direct applications regarding the improvement of the quality of software produced through automatic programming. In a broader sense, applications of formal methods in software engineering depend critically on the use of automated theorem provers to provide improved support for the development of safety critical systems. The development of correctness preserving program synthesis and optimization systems will help provide such support.

Subject area (key words): Automated Reasoning (program verification/synthesis/optimization), theorem proving, formal methods, planning).

1 Introduction

There is a growing interest in automatic theorem proving for program development (for example, synthesis, improvement, transformation and verification). Systems supporting the manipulation of non-trivial programs are complex and are at best semi-automatic.

In this extended abstract we report on novel research concerning the exploitation of synthesis proofs for the purposes of automatic program optimization and the automatic synthesis of efficient programs. Our approach has numerous advantages over more traditional approaches to program optimization which we shall address in the subsequent sections. Firstly, we set the research in context by addressing the real-world applications of this research. Secondly, we describe the automatic *transformation* of synthesis proofs yielding inefficient programs into synthesis proofs that yield efficient programs. Finally, we describe program inprovement through the automatic *synthesis* of efficient programs from the equational definitions that characterized inefficient programs.

1.1 Applications

A current dilemma in the field of Computer Science is that demands for quality and complexity of software are outstripping the tools currently available. As computer programs play an increasingly important role in all our lives so we must depend more and more on techniques for ensuring the high quality (*efficiency* and *reliability*) of computer programs. By *efficient* we mean that a program is designed to compute a task with minimum overhead and with maximum space and time efficiency. By *reliable* we mean that a program is ensured, or guaranteed in some sense, to compute the desired, or specified, task.

The most promising technique being developed for the automatic development of high quality software are *formal methods*. Applications of formal methods in software engineering depend critically on the use of automated theorem provers to provide improved support for the development of safety critical systems. Potentially catastrophic consequences can derive from the failure of computerized systems upon which human lives rely such as medical diagnostic systems, air traffic control systems and defence systems (the recent failure of the computerized system controlling the London Ambulance Service provides an example of how serious software failure can be). Formal methods are used to provide programs with, or prove that programs have, certain properties: a program may be proved to *terminate*; two programs may be proved equivalent; an inefficient program may be *transformed* into an equivalent efficient program; a program may be *verified* to satisfy some specification (i.e. a program is proved to compute the specified function/relation); and a program may be *synthesized* that satisfies some specification.

The research described herein addresses both the reliability and efficiency, as well as the automatability, aspects of developing high quality software using formal methods. We describe general theorem proving techniques for automatic program optimization and synthesis. In both cases the target program is a significant improvement on the source (efficiency), and is guaranteed to satisfy the desired program specification (reliability).

Further applications of this research include the optimization of electronic circuit design and the optimization of computer configurations. This is because both these design problems can be formally cast as processes of inference [Basin, 1991, Lowe, 1991]. Thus, we can apply the same automated theorem proving techniques that we use for high quality software production.

1.2 Background: the *Proofs as Programs Paradigm*

The problems of automated program synthesis and verification have already been addressed within the *proofs as programs paradigm* [Horn & Smaill, 1990, Constable *et al*, 1986], [Bundy *et al*, 1990c]. Constructive logic allows us to correlate computation with logical inference. This is because proofs of propositions in such a logic require us to construct objects, such as functions and sets, in a similar way that programs require that actual objects are constructed in the course of computing a procedure. Historically, this correlation is accounted for by the *Curry-Howard isomorphism* which draws a duality between the inference rules and the functional terms of the λ-calculus [Curry & Feys, 1958, Howard, 1980].

Such considerations allow us to correlate each proof of a proposition with a specific λ-term, λ-terms with programs, and the proposition with a specification of the program. Hence the task of generating a program is treated as the task of proving a theorem: by performing a proof of a formal specification expressed in constructive logic, stating the *input-output* conditions of the desired program, an algorithm can be routinely extracted from the proof. Furthermore, different constructive proofs of the same proposition correspond to different ways of computing a specific program specification. Thus by controlling the form of a specification proof we can control the efficiency with which the constructed program computes the specified goal. Here in lies the key to both synthesizing *efficient* programs, and to transforming synthesis proofs that yield inefficient programs into proofs that yield efficient programs.

2 Program Optimization by Proof Transformation

We have implemented a system for optimizing programs through the transformation of synthesis proofs [Madden, 1989, Madden, 1991, Madden, 1992]. These proofs are based on a Martin-Löf type theory logic and proved within the OYSTER proof refinement system (Martin-Löf, 1979; Martin-Löf, 1984).[2] The system has the desirable properties of *automatability*, *correctness* and mechanisms for *reducing the transformation search space*, and various *control mechanisms* for guiding search through that space.

As with synthesis and verification, knowledge of theorem proving, and in particular automatic proof guidance techniques, can be brought to bear on the transformation task. Furthermore, the proof transformations allow the human synthesizer to produce an elegant *source* proof, without clouding the theorem proving process with efficiency issues, and then to transform this into an opaque proof that yields an efficient *target* program.

To accomplish program transformation *through* proof transformation, we have successfully, and for the first time, adapted a range of program transformation techniques to the proofs as program paradigm, notably: the *tupling* technique for "merging" repeated (sub)computations, [Pettorossi, 1984] [Chin, 1990], and the fold/unfold technique for transforming inefficient functional programs into equivalent, more efficient, functional programs by a process of unfolding and folding definitions [Darlington, 1981].

Synthesis proofs differ from straightforward programs in that more information is formalized in the proof than in the program: a description, or *specification*, of the task being performed; a *verification* of the method; and an account of the *dependencies between facts involved in the computation*. Thus, synthesis proofs represent a *program design record* because they encapsulate the reasoning behind the program design by making explicit the procedural commitments and decisions made by the synthesizer. This extra information means that proofs lend themselves better to *transformation* than programs since one expects that the data relevant to the transformation of algorithms will be different and more extensive than the data needed for simple execution. In particular, dependency information abstracted from the source proof guides the transformations without the need for any extensive analysis of a programs recursive behaviour (such as the use of symbolic dependency graphs to analyse a programs recursive calling structure).

A key feature of our approach to program optimization consists in the transformation of the various induction schemas employed in OYSTER synthesis proofs.

[2] OYSTER is the Edinburgh Prolog implementation, and extension, of NUPRL; version "nu" of the *Proof Refinement Logic* system originally developed at Cornell [Bundy *et al*, 1990b],[Horn & Smaill, 1990, Constable *et al*, 1986].

Of particular importance to inducing recursion in the extracted algorithm is the employment of *mathematical induction* in the synthesis proofs: to each form of induction employed in the proof there corresponds a dual form of recursion. Such dualities offer the user a handle on the type, and efficiency, of recursive behaviour exhibited by the extracted algorithm.

The source and target programs of traditional program transformation systems do not have a formal specification present. This means there is no immediate means of checking that the target program meets the desired operational criteria. Regarding proof transformation, all transformed proofs yeild programs that are correct with respect to the specification goals. By having a specification present we also ensure that the target computes the same specified input/output relation as the source, only more efficiently.

3 Program Optimization by Proof Synthesis

In the Mathematical Reasoning Group, at the Edinburgh University Department of AI, we are involved in automating inductive theorem proving using a meta-level control paradigm called 'proof planning' [Bundy, 1988]. Our proof planning system, CLAM, is able to prove a large number of inductive theorems automatically [Bundy *et al*, 1991b]. Proof plans are formal outlines of constructive proofs and provide a means for expressing, in a meta-language, the common patterns that define a family of proofs [Bundy, *et al*, 1991a, Madden *et al*, 1993b]. A tactic expresses the structure of a proof strategy at the level of the inference rules of the object-level logic. Proof plans are constructed from the tactic specifications called *methods*. Using a meta-logic, a method captures explicitly the preconditions under which a tactic is applicable.

We are using proof plans to control the (automatic) synthesis of efficient functional programs, specified in a standard equational form, $\mathcal{E}$, by using the proofs as programs principle [Madden *et al*, 1993a]. The goal is that the program extracted from a constructive proof of the specification is an optimization of that defined solely by $\mathcal{E}$. Thus the theorem proving process is a form of program optimization allowing for the construction of an efficient, *target*, program from the definition of an inefficient, *source*, program. Our main concern lies with optimizing the recursive behaviour of programs through the use of proof plans for inductive proofs. Thus we exploit the duality between induction and recursion which forms one aspect of the *Curry-Howard isomorphism*.

We view program synthesis then as the combination of verification and *middle-out reasoning*. Middle-out reasoning is a technique that allows us to solve the typical eureka problems arising during the synthesis of efficient programs by allowing the planning to proceed even though certain object-level objects are unknown (e.g. identification of induction schema, recursive types etc.) Subsequent plan-

ning then provides the necessary information which, together with the original definitional equations, allows for the instantiation of such meta-variables through higher-order unification procedures.

More interesting however, is to introduce meta-variables into the proof to represent the unknown parts of an *improved* program (e.g., constraint functions). Proceeding in the same way, proof planning gradually instantiates these variables until a new, improved program is synthesized. Different proof plans provide different (inductive) proof structures, and hence different recursive extracts.

We have found this approach to be very promising. This is demonstrated by the success we have had in expressing a wide variety of well-known, but disparate, program improvement techniques within the proof planning framework [20]. For example, constraint-based transformation, generalization, fusion and tupling can be seen as proof planning. In addition, we have extended the work of Wadler, [Wadler, 1988] and later Chin, [Chin, 1990], to encompass a larger class of functions that can be usefully optimized using the influential *deforestation technique* [Wiggins *et al*, 1991]. We believe that middle-out reasoning will play an increasingly important role in theorem proving, since it allows us to address important problems like choosing induction schemata, and existential witnesses (which correlate to recursion schemata and recursive data types).

4 Conclusion

We have described two implemented techniques for producing high quality (efficient and reliable) software using the proofs as programs paradigm. Both the transformation and synthesis techniques satisfy the desirable properties for automatic programming systems: correctness, generality, automatability and the means to guide search through the transformation space. Outside of the automatic programming domain, this research is also applicable to such tasks as the optimization of electronic circuit design and the optimization of computer configurations. In the longer term, the research will certainly help provide improved support for the development of safety critical systems.

References

[Basin, 1991] Basin, David A. (1991). Extracting circuits from constructive proofs. Research Paper 533, Dept. of Artificial Intelligence, Edinburgh, Also appeared in Proceedings of the IFIP-IEEE International Workshop on Formal Methods in VLSI Design, Miami USA, 1991.

[Bundy, 1988] Bundy, A. (1988). The Use of Explicit Plans to Guide Inductive Proofs. In Luck, R. and Overbeek, R., (eds.), *CADE9*. Springer-Verlag.

[Bundy *et al*, 1990a] Bundy, A., Smaill, A. and Hesketh, J. (1990a). Turning eureka steps into calculations in automatic program synthesis. In Clarke, S.L.H., (ed.), *Proceedings of UK IT 90*, pages 221–6. Also available from Edinburgh as DAI Research Paper 448.

[Bundy *et al*, 1990b] Bundy, A., van Harmelen, F., Horn, C. and Smaill, A. (1990b). The Oyster-Clam system. In Stickel, M.E., (ed.), *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[Bundy *et al*, 1990c] Bundy, A., van Harmelen, F., Smaill, A. and Ireland, A. (1990c). Extensions to the rippling-out tactic for guiding inductive proofs. In Stickel, M.E., (ed.), *10th International Conference on Automated Deduction*, pages 132–146. Springer-Verlag. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 459.

[Bundy *et al*, 1991a] Bundy, A., Stevens, A., van Harmelen, F., Ireland, A. and Smaill, A. (1991a). Rippling: A heuristic for guiding inductive proofs. Research Paper 567, Dept. of Artificial Intelligence, Edinburgh, To appear in The Journal of Artificial Intelligence.

[Bundy *et al*, 1991b] Bundy, A., van Harmelen, F., Hesketh, J. and Smaill, A. (1991b). Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324. Earlier version available from Edinburgh as DAI Research Paper No 413.

Chin 990] Chin, W.N 1990). *Automatic Methods for Program Transformation.* npublished Ph.D. thesis, Imperial College.

onstable e *al.* 1986] Constable, R.L., Allen, S.F., Bromley, H.M. *et al.* (1986). *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall.

[Curry & eys, 1958] Curry, H.B. and eys, R 958). *Combinatory Logic* North-Holland.

[Darlington, 1981 Darlington, J. (August 1981). An experimental program transformation and synthesis system. *Artificial Intelligence*, 16(3):1–46.

[Hesketh *et al.* 992] Hesketh, J., Bundy, A. and Smaill, A. (June 1992). Using middle-out reasoning to control the synthesis of tail-recursive programs. In Kapur, D., (ed.), *11th Conference on Automated Deduction*, pages 310–324, Saratoga Springs, NY, USA. Published as Springer Lecture Notes in Artificial Intelligence, No 607.

[Horn & Smaill, 1990] Horn, C. and Smaill, A. (1990). Theorem proving with Oyster. Research Paper 505, Dept. of Artificial Intelligence, Edinburgh, To appear in Procs IMA Unified Computation Laboratory, Stirling.

[Howard, 1980] Howard, W.A. (1980). The formulae-as-types notion of construction. In Seldin, J.P. and Hindley, J.R., (eds.), *To H.B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press.

[Lowe, 1991 Lowe, Helen. (May 1991). The use of theorem proving techniques in expert systems for configuration. In Rault, J.-C., (ed.), *Proceedings of the Eleventh International Workshop on Expert Systems and their Applications, Avignon.* EC2. Also available from Edinburgh as DAI Research Paper 536.

[Madden, 1989] Madden, P. (1989). The specialization and transformation of constructive existence proofs. In Sridharan, N.S., (ed.), *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.* Morgan Kaufmann.

[Madden, 1991] Madden, P. (1991). *Automated Program Transformation Through Proof Transformation.* Unpublished Ph.D. thesis, University of Edinburgh.

Madden, 1992] Madden, P. (1992). Automatic Program Optimization Through Proof Transformation. In Kapur, D., (ed.), *CADE11*, pages 446–461. Springer Verlag. Lecture Notes in Computer Science No. 607. Also available from Edinburgh as DAI Research Paper 604.

[Madden *et al*, 1993a] Madden, P., Green, I. and Bundy, A. (1993a). A General Technique for Optimizing Programs by using Proof Plans. Technical Report in preperation, Dept. of Artificial Intelligence, University of Edinburgh.

[Madden *et al*, 1993b] Madden, P., Hesketh, J., Green, I. and Bundy, A. (1993b). A general technique for automatically optimizing programs through the use of proof plans. Research Paper 608, Dept. of Artificial Intelligence, Edinburgh, To appear in Proceedings of LOPSTR-93.

[Pettorossi, 1984] Pettorossi, A. (1984). A powerfull strategy for deriving programs by transformation. In *ACM Lisp and Functional Programming Conference*, pages 405–426.

[Wadler, 1988] Wadler, P. (1988). Deforestation: Transforming Programs to Eliminate Trees. In *Proceedings of European Symposium on Programming*, pages 344–358. Nancy, France.

[Wiggins *et al*, 1991] Wiggins, G. A., Bundy, A., Kraan, I. and Hesketh, J. (1991). Synthesis and transformation of logic programs through constructive, inductive proof. In Lau, K-K. and Clement, T., (eds.), *Proceedings of LoPSTr-91*, pages 27–45. Springer Verlag. Workshops in Computing Series.